

VIDAS: Object-based Virtualized Data Sharing for High Performance Storage I/O

Pablo Llopis Javier Garcia Blas Florin Isaila
pllopis@arcos.inf.uc3m.es fjblas@arcos.inf.uc3m.es florin@arcos.inf.uc3m.es

Jesus Carretero
jcarrete@arcos.inf.uc3m.es

Computer Architecture and Technology Area
Universidad Carlos III de Madrid
Madrid, Spain

ABSTRACT

With scientific computing in the cloud gaining popularity and using every time larger data sets, high performance storage I/O in virtualized environments is substantially increasing in importance. However, exploiting the performance potential of the storage I/O on today's virtualized architectures is complex, due to the limitations of POSIX standard for storage I/O and the lack of integration of related mechanisms such as data sharing, storage I/O coordination, relaxing the consistency semantics, and data locality awareness.

In this paper we propose VIDAS (Virtualized DATA Sharing), an object-based virtualized data store that targets to integrate the above mechanisms through a simple and powerful interface. VIDAS can be used to efficiently and consistently share access to externally stored data in virtualized environments based on a shared pool of storage objects. We show how VIDAS can be used for straightforwardly implementing I/O coordination and data sharing for two common high-performance patterns: inter-domain write-reader and inter-domain collective I/O. We present the implementation and evaluation of VIDAS for the Xen virtualization solution. In addition, we present a novel mechanism for efficiently sharing memory among an arbitrary number of virtual machines.

Categories and Subject Descriptors

D.4.2 [Operating Systems]: Storage Management—*Storage hierarchies, Virtual memory*; D.4.3 [Operating Systems]: File Systems Management—*Access methods, Distributed file systems*

Keywords

HPC; cloud computing, storage I/O virtualization, data sharing

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ScienceCloud'13, June 17, 2013, New York, NY, USA.
Copyright 2013 ACM 978-1-4503-1979-9/13/06 ...\$15.00.

1. INTRODUCTION

In the last years the clouds based on virtualized architectures have become increasingly attractive for running scientific codes. On one hand clouds offer cost- and resource-efficient solutions due to on-demand scalability and pay-per use model. On the other hand virtualization technologies provide several advantages including flexibility to run customized versions of the operating system, performance isolation of workloads, improved security and reliability, migration for load balancing and fault tolerance.

One of the most important current challenges for broadening the adoption of virtualized cloud solutions by the HPC community is providing efficient access to data-intensive scientific applications. Scientific applications require both high-performance storage I/O and efficient data sharing solutions. Various options exist for data sharing in a virtualized cloud environment, including distributed and parallel file systems, object-based storage systems, and databases [4]. However, there are still several challenges, which have to be faced for exploiting the full performance and scalability potential of the cloud resources [15]. One of the main current challenges is how to address the limitation of the still popular POSIX file system interface. There is a wide agreement that POSIX consistency model is not suitable for high-performance and scalable applications. Another critique of POSIX is that it does not expose data locality, while researchers agree that locality-awareness is a key factor for building high performance scalable systems [13]. These issues are magnified in virtualized environments, one of the reasons being the trade off between protection and performance. Protection across domains is enforced at the cost of memory copy operations, which degrade the performance. Cooperating applications (e.g. on-line visualization of a scientific application) or processes of a single application (e.g. workflow application) running in several virtualized domains are currently offered few possibilities to coordinate the storage I/O across domains and to trade off performance and protection.

Our main contribution in this paper is to address this gap by proposing a set of abstractions and mechanisms which enable building efficient virtualized storage systems for data-sharing applications, while trading off protection and performance. More precisely, we propose abstractions and mechanisms, which allow to:

- Coordinate storage I/O across domains.
- Create shared access spaces across node-local domains.

- Relax the POSIX consistency.
- Allow for flexible data write and data update policies.
- Expose data locality.

VIDAS is built on top of a new mechanism for collectively sharing memory among multiple virtual machines.

The remainder of this paper has the following structure. Section 2 presents related work on storage I/O virtualization. Section 3 introduces the novel abstractions and mechanisms for data sharing in virtualized environments. Section 4 presents the implementation. Section 5 describes two use cases. Section 6 evaluates our solution. Finally, Section 7 concludes.

2. RELATED WORK

This section describes existing storage I/O virtualization solutions for environments running virtual machines as depicted in Figure 1. In these environments, it is common that a virtualization software layer called *hypervisor* multiplexes host physical resources among several virtual machines running *guest* operating systems. Common storage I/O virtualization solutions can be broadly classified in two categories: block-level virtualization (labeled in the figure A1, A2, and A3) and filesystem-level virtualization (labeled in the figure B1, B2, B3).

A block-level device can be fully virtualized either on top of a physical device driver (A1) or on top of a file system (A2). These solutions emulate a device driver in the host and have the advantage of flexibility (a virtual disk can be mapped to any physical device), but suffer performance penalties due to duplicated functionality. Most virtualization platforms support emulated devices for backwards compatibility, while offering paravirtualized device drivers for high performance [18]. Paravirtualized device drivers (A3) eliminate the need for duplicated device drivers. In most modern virtualization platforms their implementation is split into a front-end running in a guest and a back-end serving front-end requests and running the device drivers natively. Paravirtualized drivers can be used for solutions A1, A2, B1, and B2, in order to reduce device emulation overhead in the guest.

Paravirtualized device drivers are available for most virtualization solutions, such as KVM virtio drivers [14], Xen split drivers [12], and VMWare’s guest disk driver. Parallax [9] is a related block-level virtualization solution that provides sharing of virtual disk images (VDI). In Parallax a device driver backend can be a remote host. Parallax does not support write-sharing of VDIs. In turn, it offers efficient operations for fine-grained frequent snapshotting of VDIs.

Guest file systems can be either purely virtualized or paravirtualized. A purely virtualized solution can be based on A1, by uniquely assigning a physical disk to an emulated device (not shown in the figure). Alternatively, nested file systems (B1) consist of a guest file system running on top of a host file systems. This approach entails redundant functionality which results in performance penalties: interrupts, duplicated and uncoordinated disk schedulers, redundant memory copies, duplicated buffer caches [6]. An additional drawback is that applications running on the guest domain do not have any control over the data flow through all storage layers.

A *paravirtualized file system* file system solution can be simply based on running a unmodified file system on a paravirtualized device driver as shown in B2. However, more

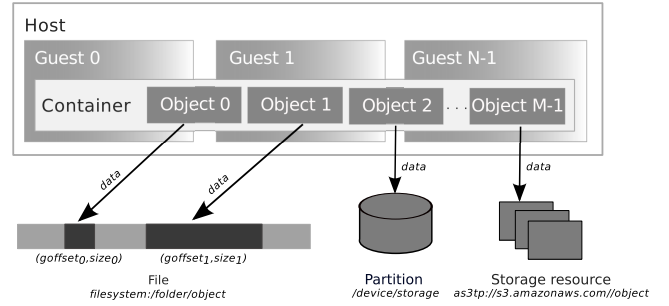


Figure 2: Several guests sharing objects through a container. Each object is mapped on an external storage resource.

complex paravirtualized file system solutions (B3) [10, 11] allow a stronger inter-domain coordination in order to optimize away redundant functionality.

VIDAS, as a paravirtualized object-level storage pool, can not be straightforwardly categorized in this classification. We see VIDAS as an intermediary layer between disk-level virtualization and file system-level virtualization. On one hand, VIDAS can be used to consistently share access to a non-shared disk. On the other hand, VIDAS can serve as an intermediary layer for building a shared paravirtualized file system by allowing a straightforward implementation of a shared name space or a shared buffer cache on top of storage objects. In general, VIDAS can be used for offering guests high-performance data sharing and locality awareness based on a shared pool of storage objects. The shared pool consists of objects that can be mapped to every virtual machine running on the same host. Our memory sharing mechanism differs from existing mechanisms for communicating and data sharing between virtual machines such as XenSocket [19], XWAY [5], and others [7, 3], which typically share a page between two domains. Our solution supports the capability of collectively sharing data among an arbitrary number of domains, as detailed in Section 3.

3. ABSTRACTIONS AND MECHANISMS FOR VIRTUALIZED DATA SHARING

This section introduces the novel abstractions and mechanisms for data sharing in virtualized environments. Our virtualized data sharing solution is based on two abstractions: containers and storage objects (as shown in Figure 2). A *container* (described in Section 3.1) is an abstraction which allows data sharing between virtual domains running on the same physical node. The data sharing can be done at the granularity of a data *storage object* (described in Section 3.2). All domains with access to a common container can use it to share data objects among each other. Storage I/O coordination, data sharing, asynchronous I/O can be done at object-level.

3.1 Containers

A container in VIDAS is an abstraction which facilitates storage object sharing across virtual domains. A container has a unique name, which has to be known by the domains who want to share it. VIDAS provides a restricted set of container operations as listed in Table 1.

A container is created by an initiator domain by specifying

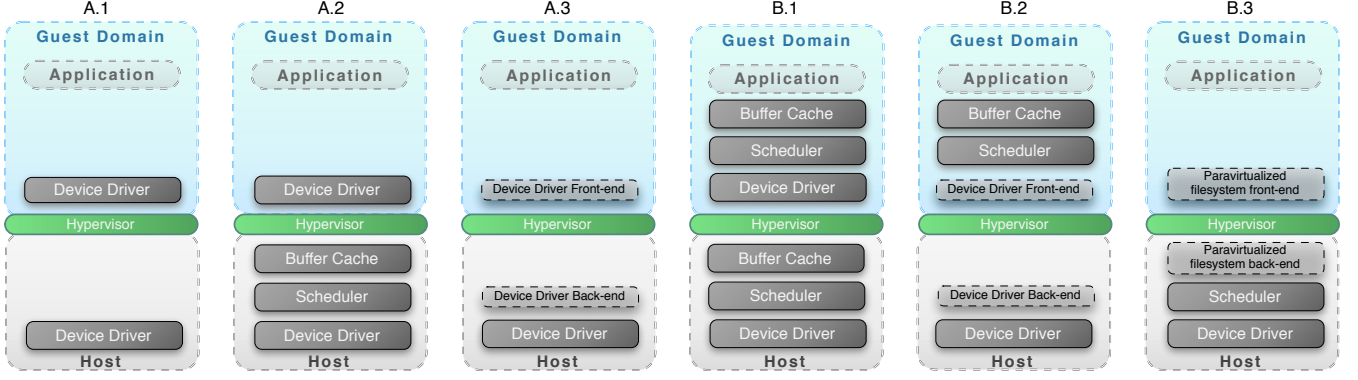


Figure 1: Storage virtualization solutions. The solutions labeled A depict block-level device virtualization, while solutions labeled B depict filesystem-level virtualization.

Table 1: List of container operations.
Container operations

```
int container_create(char *name, int domain_ids[])
int container_destroy(char* name)
int container_attach(char *name)
int container_leave(char* name)
```

ing a unique name and an array of potential domains that are allowed to share it. A container can be destroyed by the initiator domain only if there is no other domain sharing it. A domain different from the initiator can share a container by calling `container_attach` and unshare it by calling `container_leave`. After sharing a container, domains are able to start to manipulate and access storage objects shared by any one of them as described in the next section.

3.2 Storage objects

A storage object in VIDAS is an abstraction for data sharing across domains. Each storage object is uniquely associated to an external storage resource through its name. The external storage resource can be a file from a file system, an object from a storage system, a disk partition, or any other storage resource that can be uniquely identified through a name and offers a linear address space. For instance, the external storage resource can be a file from a locally mounted NFS or a URL of a remote object stored in Amazon S3. In this paper we will assume that a simple get/put interface is available for accessing these external storage resources and we will concentrate on the node-level data sharing in a virtualized environment.

The storage objects are different from traditional POSIX files in several aspects:

- Each object is associated with a user-extensible list of name-value attributes.
- Strong consistency is not enforced, but optional.
- Data writes to external storage resources can be guided by a configurable policy such as write-through or write-back.
- Applications can learn if object data is cached in memory, providing locality awareness.
- Operations on objects are stateless.

Table 2: List of object operations.
Object metadata operations

```
obj_handle_t object_create(char* ext_storage_rsc,
                           size_t offset, size_t size, char* cname)
obj_handle_t object_join(char* ext_storage_rsc,
                          size_t offset, size_t size, char* cname)
int object_get_locality(char* ext_storage_rsc,
                       obj_handle_t *objects[])
int object_leave(obj_handle_t o)
int object_destroy(obj_handle_t o)
int object_getattr(obj_handle_t o, char *name,
                  void *value, size_t size)
int object_setattr(obj_handle_t o, char *name,
                  void *value, size_t size)
```

Object data access operations

```
int obj_write(obj_handle_t o, char *buf, size_t offset, size_t sz)
int obj_read(obj_handle_t o, char *buf, size_t offset, size_t sz)
int obj_flush(obj_handle_t o)
int obj_update(obj_handle_t o)
```

Object synchronization operations

```
int object_wait(obj_handle_t o, char **bufp)
int object_notify(obj_handle_t o, char *buf)
```

In the remainder of this section we discuss storage object operations (shown in Table 2).

3.2.1 Metadata operations

Each VIDAS storage object can be uniquely associated to an external storage resource through the `obj_create` call. After creation, any other domain can share the object by joining through the common container (`obj_join`). The `obj_create` operation associates a data object to a container, i.e. reserves shared memory in a container for the given data range of an object. Domain access control is enforced through the container associated to the object, which specifies in which domains the object is accessible. Object attributes can be manipulated using the `object_getattr` and `object_setattr` operations. Predefined attributes include: “name” (the external storage resource name, i.e. a file or an URL), “data” (a pointer to the object’s data), “offset”

(the offset where the object maps on to the storage destination), “size” (size of the region of the data item shared by this object), “container” (container through which the object is shared), “synchronized” (if “true”, the operations on this object are atomic), “write policy” and “read policy” (which are further discussed in the next subsection). Further attributes can be defined by users through `object_setattr`.

3.2.2 Data operations

After an object has been shared by a domain (either through `obj_create` or `obj_join`), it can be accessed from the domain through standard `obj_read` / `obj_write` calls. Alternatively, the pointer to the object data can be retrieved through an `object_getattr` operation on the “data” attribute and, subsequently, directly accessed. The first alternative offers “opaque” access to the object and can be used together with the attribute “synchronized” in order to provide atomic access to the object. In the atomic mode (the attribute “synchronized” has the value “true”), accesses to non-overlapping object intervals can proceed concurrently. The second alternative provides a “zero-copy” mode, for which object modifications from any domains become instantaneously visible to all the other domains. However, in this mode the user is responsible to enforce access consistency.

The “update policy” and “write policy” object attributes reflect how the data flows between a VIDAS object and the represented external storage resource. The “update policy” decides if external object updates are applied “lazy” or “eager”. For lazy updates the object is updated from the external storage resource, only when the user calls `object_update`. The “write policy” refers to how object modifications are propagated to the external storage resource. If the write policy is “write-through”, object modifications are propagated right away. For the “write-back policy”, the object modifications are propagated only when the user calls `object_flush`.

3.2.3 Synchronization operations

As discussed in the previous subsection, atomic access to an object is provided when the “synchronized” attribute is set to “true” and the `obj_read` / `obj_write` are used. For coordinating the access to objects from different domains, VIDAS provides a wait/notify mechanism. A notify operation `obj_notify` sends a message to an object. A different domain can block waiting to receive the message for the given object by calling `obj_wait`. Section 5 shows an example of using wait/notify for implementing a collective I/O write operation.

4. IMPLEMENTATION

We have implemented VIDAS interface based on Xen virtualization solution [12]. This section describes Xen implementation details. In order to simplify the reading, we start by presenting the Xen inter-domain mechanisms leveraged by our implementation. Subsequently, we present the implementation of the multi-domain data sharing mechanism. Finally, we discuss container and object implementations.

4.1 Xen inter-domain mechanisms

VIDAS implementation leverages three main inter-domain Xen mechanisms: XenStore, shared memory, and ring buffers [12].

XenStore. XenStore is a key-value centralized data base,

which is shared by all Xen domains and is typically used for passing configuration parameters across domains.

Xen shared memory. The Xen mechanism for sharing the memory is based on a mechanism called grant table. Each domain has its own grant table which is shared with Xen. Each entry of a grant table informs Xen about the pages shared by the owning domain with other domains. Each grant table is indexed by a grant reference. A domain dom_i performs the following actions in order to share pages with dom_j . First, it calls a function (`grant_foreign_access`) with two parameters: the target domain dom_j and the number of pages to share with dom_j . This function allocates memory, assigns it to a grant table entry, and returns a grant reference and a local index. The grant reference is passed over XenStore to dom_j , while the local index is used by `mmap` to map the pages to the virtual memory space of dom_i . On its turn, dom_j retrieves the reference grant from XenStore, calls an `ioctl` function using the grant reference in order to retrieve the local index, and, finally, uses the local index to map the pages to its virtual memory space through a `mmap` call. At this moment the pages are shared between domains dom_i and dom_j .

Xen ring buffers. For inter-domain communication, a producer/consumer circular queue known as a ring buffer, is implemented on top of a shared memory buffer. This ring buffer acts as the transport mechanism between domains for implementing inter-domain communication.

4.2 Xen inter-domain mechanisms in VIDAS

VIDAS implementation leverages shared memory for sharing object data, attributes, and other opaque metadata across used domains and ring buffers for communication between the user domains and the host. Figure 3 depicts an overview of our implementation, which shows how ring buffers and shared memory are used in VIDAS.

The memory is shared between two domains based on the procedure described in the previous subsection. For sharing a page among n domains, in our solution the domain initiating the sharing inserts in its grant table $n - 1$ entries, all of which are associated with the same page. Subsequently, each of the other $n - 1$ domains receives a different grant reference, representing the same physical page. The page is then mapped as in a two page case¹.

The ring buffers are used for implementing a lightweight Remote Procedure Call (RPC) based on a front-end, running on the calling domain, and a back-end running on the host. The back-end waits on the ring buffer to receive call messages from the front-end, performs the call, and returns the result in the ring buffer. After performing the call, the front-end waits for the result from the back-end. In this work both the back-end and front-end use polling, while an interrupt based approach was left for the future work. Our initial choice was based on the conclusion of a study, which showed that the use of polling in optimizing the storage virtualization can substantially reduce the overhead of interrupt handling [1].

4.3 Container implementation

The container management is performed at the host. All container operations are implemented as lightweight RPCs, as described in the previous subsection. When creating a

¹We provide this new Xen feature at <https://github.com/pllopolis/linux/commit/fb6dca>

container, the guest forwards the call to the host, which stores the domain IDs sharing access. A subsequent container attach operation is successful only if the calling domain belongs to the list specified by the creating domain. Destroying a container and leaving from a container are simple RPCs that remove or update the container metadata from the host.

4.4 Object implementation

Object management is also performed at the host. However, most operations on object data and attributes do not involve the host, as described below.

Seven operations from VIDAS interface rely on RPCs to the host. In order to create an object (`object_create`), a domain follows the steps described above: allocates memory for object data, object attributes and other object metadata, is assigned a grant reference, passes the grant reference to all memory sharing domains, and maps the page to its virtual memory. Finally, it contacts the host with an RPC and informs about the newly created object. At this point each sharing remote domain is entitled to share the object by calling `object_attach`, which maps the object memory to the remote domain virtual memory and informs the host through an RPC. The operations `object_destroy` and `object_leave` undo the create and attach operations, respectively, and inform the host to remove the object from the index. The function `object_get_locality` is implemented as an RPC which returns from the host all the local objects (created or attached) associated to the external storage resource. The other two operations, whose implementation leverages RPCs to the host are `object_flush` and `object_update`, which simply ask the host to flush/update the object to/from the remote storage resource.

All the other six VIDAS operations rely on shared memory and do not directly involve the host. As the object attributes are stored in main memory, the functions `object_getattr` and `object_setattr` directly work on the shared memory, after implicitly taking a mutex in order to ensure consistency. The mutex is implemented by using Linux atomic test-and-set operations². The data access operations `object_write` and `object_read` access the shared memory directly and are atomic only if the *synchronized* object attribute is set to “true”. In VIDAS, accesses to an object are serialized only if the accessed domains overlap. For addressing this issue in our current implementation, the object metadata includes an array of mutexes which provide mutual exclusion to overlapping access domains. A further extension to this implementation for providing multiple-reader one-writer access is straightforward.

Finally, `object_notify` and `object_wait` work as well directly on a buffer shared across the domains when object is created/attached. The `object_notify` operation copies the message to the shared memory and atomically modifies a producer pointer, while `object_wait` retrieves the message when available and passes it to the calling domain. We use polling for `object_wait`, while leaving a blocking version for future work and evaluation.

While there are numerous ways to design cooperative data sharing, we made an effort to minimize host intervention where it was not strictly needed. This is an important design decision because context switches result in hypervisor

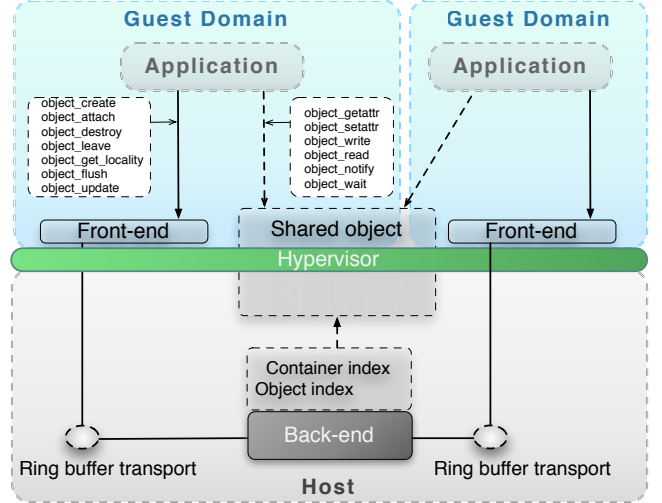


Figure 3: VIDAS implementation. Seven operations rely on RPCs to the host implemented with Xen ring buffers. All the other six VIDAS operations rely on shared memory and do not directly involve the host.

exit operations, which are known to be the main cause of performance overheads for virtualized I/O-intensive workloads [2]. Therefore, operations which manipulate object data and metadata are carried out with minimal context switches. For other operations such as object creation and removal (which are still very fast, as demonstrated in the evaluation), we chose to sacrifice a context switch for consistency and simplicity, by having a single copy of container and object indexes maintained by the host.

5. USE CASES

In this section we present the implementation of two use cases based on VIDAS interface and abstractions: inter-domain write-read sharing of a file (Subsection 5.1) and inter-domain collective I/O (Subsection 5.2). These two use cases are part of the evaluation in Section 6.

5.1 Inter-domain write and read sharing

A significant class of scientific applications is based on workflows, in which the output of one process is the input of the next [17]. While processes communicating through files within the same host can take advantage of data locality through the buffer cache, processes running within the same host but on different domains require efficient inter-domain data sharing solutions. In this section we show how the building block of a write-read pattern of a workflow can be simply and efficiently implemented based on VIDAS.

Listing 1 shows the implementation. We assume that the writer executes in domain *i* and the reader in domain *j*. Before calling the I/O storage functions write or read, domain *i* creates a container “con” and permits domain *j* to share it, while domain *j* joins this container. Subsequently, domain *i* creates an object of *size* bytes mapped to the offset *offset* of the external storage resource *ext_storage_rsc* (a file or a URL), sets the attribute *synchronized* to “true” (in order to enable atomic accesses), writes the data, and notifies the object modification. Domain *j* joins the object, waits for a notification, and reads the data. In the listing the reading

²<http://old-list-archives.xen.org/xen-devel/2009-03/msg01823.html>

Write operation executed in domain i

```
// Share the object with domain j
int domain_ids[]={j};
// Create the container
container_create("con", domain_ids);

write(char* ext_storage_rsc, char* buf, size_t
    offset, size_t size) {
    // Create the storage object
    object_handler_t o = object_create(
        ext_storage_rsc, offset, size, "con");
    // Indicate the all accesses will be atomic
    object_setattr(obj, "synchronized", "true",
        5);
    // Write the data to objects in segments of
        size b
    object_write(o, buf, 0, size);
    // Notify that the data is available
    object_notify(o, NULL);
    // Destroy the object
    object_destroy(o);
}
// Destroy the container
container_destroy("con");
```

Read operation executed in domain j

```
// Attach to an existing container
container_attach("con");

read(char* ext_storage_rsc, char* buf, size_t
    offset, size_t size) {
    // Join the object to mapping to a common
        external storage resource
    object_handler_t o = object_join(ext_storage_rsc,
        offset, size, "con");
    // Wait for the data to become available
    object_wait(o, NULL);
    // Read the data to objects in segments of size
        b
    object_read(o, buf, 0, size);
    // Unmap the object
    object_leave(o);
}
// Leave the container
container_leave("con");
```

Listing 1: Inter-domain write-read data sharing.

of the data is done through the explicit *obj_read* operation. However, it is possible to avoid copying the data involved in this operation, by simply retrieving the pointer to the shared buffer through the “data” attribute and directly use the data.

The way the data modifications propagate from the shared object to the external storage resource can be controlled through the **write policy** attribute (not shown in Listing 1).

The write/read operations presented above can be straightforwardly used as a building block for a producer-consumer implementation or for a data streaming implementation.

5.2 Inter-domain collective I/O

I/O intensive applications often face the problem of accessing non-contiguous portions of data. In scientific applications it is often the case that while different processes access non-contiguous portions of data, requests of a group of processes may together span a contiguous portion [16]. The optimizations merging different requests from cooper-

Write is called from domains d_0, d_1, \dots, d_{n-1}

We assume domain d_0 will be the aggregator

```
write(char* ext_storage_rsc, char* bufs[], size_t
    offsets[], size_t sizes[]) {
    object_handler_t o;
    // Get my domain and create/share the object
    int my_domain = get_domain();
    if (my_domain == d0)
        o = object_create(ext_storage_rsc, offset,
            size, "con");
    else
        o = object_join(ext_storage_rsc, offset, size,
            "con");
    // Each domain writes data to the object by
        calling
    // several times object_write on the object
        o
    ...
    // Each domain notifies the modifications
    object_notify(o, NULL);
    // Domain d0 waits for all notifications before
        flushing the data
    if (my_domain == d0) {
        for (i=0; i<n; i++)
            object_wait(o, NULL);
        object_flush(o);
    }
}
```

Listing 2: Inter-domain collective I/O write implemetation.

ating processes into a single large I/O operation are referred to as collective I/O [16].

In a purely virtualized environment, efficient collective I/O is difficult to achieve because domains are isolated from each other and data has to be shared through a network file system or network communication protocols. However, VIDAS abstractions and mechanisms allow for efficiently sharing data and coordinating accesses, as shown in Listing 2. In this case study domains d_0, d_1, \dots , and d_{n-1} write non-contiguous pieces of data of sizes given by the *sizes* vector from the buffers *bufs* to the external storage resource *ext_storage_rsc* at several offsets given by the *offsets* vector. We assume that the container “con” has been already created and shared and that the **write policy** attribute is set to “back”, i.e. write back. The implementation uses a function *get_domain* that returns a unique domain name (in Xen we implemented this function by simply requesting this value from XenStore).

Domain d_0 creates an object and all the other domains are sharing it. Subsequently, all domains write the data to the shared object through simple memory copy operations. Subsequently, all n domains notify the object that they have performed the modifications. Domain d_0 waits for all notifications to arrive (for simplicity we show as well the notification sent by d_0 to itself, but this can be obviously optimized away) and, subsequently, flushes the modifications to the external storage resource. Collective read operations can be implemented in a similar fashion.

6. EVALUATION

We evaluated the VIDAS prototype on a 12-core Intel(R) Xeon(R) CPU E5-2620 @ 2.0Ghz with 64GB of DDR3 synchronous memory clocked at 1333Mhz. The hard disk is

a Toshiba MK1002TS with a capacity of 1TB, a speed of 7200 rpm, a 64MB cache, and is partitioned with LVM and ext4. Xen version 4.2 runs Linux 3.5.7 as Dom0, modified to support sharing memory pages across an arbitrary number of domains. Preceding each experiment, we cleared caches, directory entries and inodes from memory using the Linux `drop_caches` interface.

6.1 Object operations

In this section we present an evaluation of VIDAS object and container operations. The container operations take between $64\mu s$ and $70\mu s$, which correspond to the time of the RPC between the guest and host domains. The creation of an object of one 4096 byte page takes $207\mu s$, representing the time to allocate memory, send the grant reference to the remote domain, map the page, and perform an RPC for registering the domain. Joining an object takes $98\mu s$, corresponding to the time to map the page and register at the host through a RPC. Leaving an object involves an unmap operation and an RPC and takes $82\mu s$. The other operations involve shared memory and take $2\mu s$ (setting and getting an attribute of 1 byte) and $6\mu s$ (notify/wait a message of 64 bytes).

6.2 Inter-domain communication

Due to the fact that the goal of this work is to improve data sharing in virtualized environments, we first evaluate data communication between virtual machines without performing I/O. We evaluate broadcasting 128MB data objects to 2, 4, 8 and 16 virtual machines using existing inter-domain memory sharing solutions such as Xen ringbuffers, the OSU broadcast benchmark, MPI broadcast, and our solution. Figure 4 shows the effectiveness of our multi-domain memory sharing mechanism. Because the other inter-domain communicators are restricted by Xen’s current memory sharing mechanism, which limits the amount of domains sharing a page to 2, they are required to do additional memory copies. Our solution requires a single memory copy and scales better. For fairness, we introduced an additional memory copy per virtual machine in order to obtain different copies of the data object, as opposed to just one shared copy. This is also the reason why performance drops slightly when scaling the number of virtual machines.

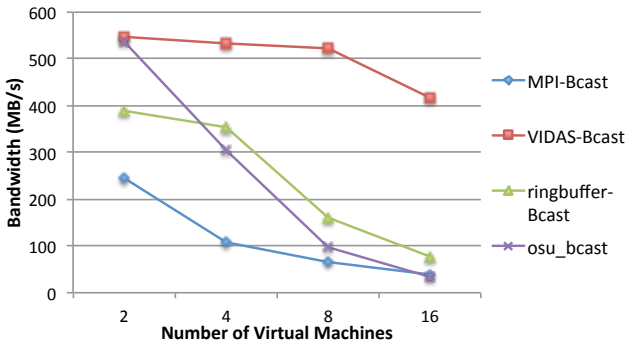


Figure 4: Evaluation of broadcast communication in VIDAS, MPI, OSU benchmark, and Xen ringbuffers.

6.3 Write and read sharing

In this subsection we compare the writer-reader implemented in VIDAS (described in 5.1) with a writer-reader based on PVFS2 and NFS using 512 MB files. VIDAS semantics enables us to perform an asynchronous write to disk while readers read object data directly from memory, thus obtaining memory read speeds, while NFS and PVFS2 rely on the disk write buffers for performing reads. We obtained a sustained throughput of over 500MB/s for VIDAS, while NFS and PVFS2 performed at close to 140MB/s.

6.4 Independent shared file read

We also evaluated shared file read, scaling the number of domains which perform overlapping reads of a 512MB file concurrently. VIDAS uses the broadcasting mechanism evaluated in Section 6.2. Figure 5 shows that VIDAS outperforms PVFS2 and NFS. VIDAS reads data into a shared object and does not require additional memory copies to transfer the data to every other domain. However, for fairness we performed an additional copy in order for each domain to have a different copy. Without this additional copy, the throughput corresponding to a disk file read would have been sustained for any number of virtual machines.

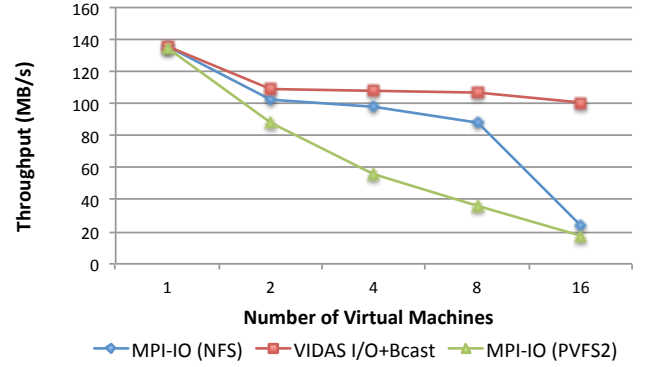


Figure 5: Evaluation of multiple reader in VIDAS, NFS, and PVFS2.

6.5 Collective I/O

In this section we compare the VIDAS inter-domain collective I/O operations (described in Subsection 5.2) with a standard collective I/O implementation from ROMIO, the most popular MPI-IO distribution. The collective I/O implementation of ROMIO is based on two-phase I/O [16], an optimization which merges non-contiguous I/O requests into contiguous ones at aggregator processes before sending them to file system (we have employed one aggregator). We have used ROMIO included in the MPICH2 1.4.1 MPI distribution. Figure 6 depicts the results for collectively writing and reading data to/from an object/file of 512 MB. The domains are accessing non-overlappingly interleaved strided vectors of 2MB blocks. Figure 6 shows the results for VIDAS collective I/O and ROMIO collective I/O implementations for reading and writing, respectively. We note that VIDAS collectives outperform collective I/O ROMIO operations. The main explanation for the better performance of VIDAS is the shared collective buffer, which helps avoid copy operations. On the other hand, ROMIO collective operations

copy the data into collective buffers before sending them to disks, which makes performance drop dramatically when increasing the number of virtual machines.

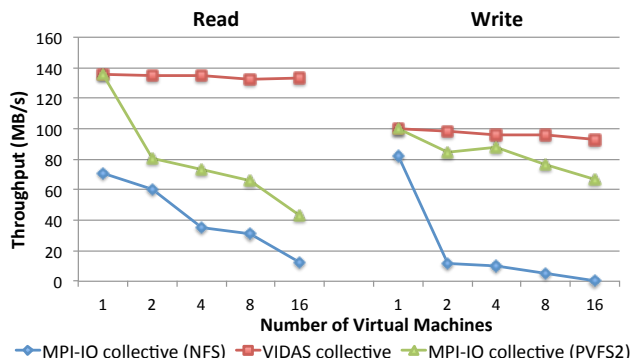


Figure 6: Comparison of VIDAS collective I/O and ROMIO collective I/O

7. CONCLUSIONS

In this paper we provide the design, implementation, and evaluation of VIDAS, an object-based virtualized data store that can be used to efficiently and consistently share access to externally stored data in virtualized environments based on a shared pool of storage objects. VIDAS provides integrated abstractions and mechanisms that allow to co-ordinate storage I/O across domains, create shared access spaces across node-local domains, relax the POSIX consistency, control the write and update policies, and control data locality. In order to efficiently implement VIDAS virtualized data sharing abstractions, we have proposed and evaluated a new data sharing mechanism which extends Xen to provide multi-domain memory sharing to user-space applications. The mechanisms and abstractions shown in this work would be best complemented with a higher-level layer which controls placement of VM machines, computing jobs, and data distribution. This would enable cloud-based HPC workloads to share data much more effectively by using VIDAS. In the future we plan to integrate VIDAS with our solution for I/O forwarding for cloud environments in order to combine node-local data sharing capabilities with high performance inter-node I/O delegation [8]. This approach would be useful for hierarchical data distribution policies based on reducing node-local communication and balancing storage I/O load over several nodes.

Acknowledgements

This work has been partially supported by the Spanish Ministry of Science under the grant IPT-430000-2010-14.

8. REFERENCES

- [1] M. Ben-Yehuda, M. Factor, E. Rom, A. Traeger, E. Borovik, and B.-A. Yassour. Adding advanced storage controller functionality via low-overhead virtualization. *FAST'12*, pages 15–15, 2012.
- [2] A. Gordon, N. Amit, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafir. ELI: bare-metal performance for I/O virtualization. In *ASPLOS '12*, pages 411–422, Mar. 2012.
- [3] W. Huang, M. J. Koop, Q. Gao, and D. K. Panda. Virtual machine aware communication libraries for high performance computing. In *SC '07*, page 9, Nov. 2007.
- [4] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling. Data Sharing Options for Scientific Workflows on Amazon EC2. pages 1–9, 2010.
- [5] K. Kim, C. Kim, S.-I. Jung, H.-S. Shin, and J.-S. Kim. Inter-domain socket communications supporting high performance and full binary compatibility on Xen. In *VEE '08*, pages 11–20. ACM, 2008.
- [6] D. Le, H. Huang, and H. Wang. Understanding performance implications of nested file systems in a virtualized environment. pages 8–8, 2012.
- [7] D. Li, H. Jin, Y. Shao, and X. Liao. A high-efficient inter-domain data transferring system for virtual machines. In *ICUIMC '09*, pages 385–390, 2009.
- [8] P. Llopis, G. Martín, B. Bergua, and J. Carretero. Virtual I/O forwarding for cloud-based HPC applications. In *ISPA '12*, pages 1–2, Apr. 2012.
- [9] D. T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M. J. Feeley, N. C. Hutchinson, and A. Warfield. Parallax: virtual disks for virtual machines. In *Eurosys '08*, pages 41–54. ACM, 2008.
- [10] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: getting beyond the limitations of virtual disks. In *NSDI '06*, 2006.
- [11] B. Pfaff, T. Garfinkel, and M. Rosenblum. Virtualization aware file systems: getting beyond the limitations of virtual disks. pages 26–26, 2006.
- [12] I. Pratt, K. Fraser, S. Hand, C. Limpach, A. Warfield, D. Magenheimer, J. Nakajima, and A. Mallick. Xen 3.0 and the Art of Virtualization. In *Proceedings of the 2005 Ottawa Linux Symposium*, July 2005.
- [13] I. Raicu, I. T. Foster, and P. Beckman. Making a case for distributed file systems at Exascale. In *HPDC '11*, pages 11–18. ACM, 2011.
- [14] R. Russell. Virtio: towards a de-facto standard for virtual I/O devices. *ACM SIGOPS Operating Systems Review*, 42(5):95–103, 2008.
- [15] J. Shafer. I/O virtualization bottlenecks in cloud computing today. *WIOV'10*, pages 5–5, 2010.
- [16] R. Thakur, W. Gropp, and E. Lusk. Data Sieving and Collective I/O in ROMIO. *FRONTIERS '99*, pages 182–. IEEE Computer Society, 1999.
- [17] E. Vairavanathan, S. Al-Kiswani, L. Costa, Z. Zhang, D. Katz, M. Wilde, and M. Ripeanu. A Workflow-Aware Storage System: An Opportunity Study. *CCGRID '12*, pages 326–334, 2012.
- [18] C. Waldspurger and M. Rosenblum. I/O virtualization. *Communications of the ACM*, 55(1), Jan. 2012.
- [19] X. Zhang, S. McIntosh, P. Rohatgi, and J. Griffin. Xensocket: A high-throughput interdomain transport for virtual machines. pages 184–203. Springer, 2007.